UNIT-II
A8601

PRESENTED BY

M.YGANDHAR
Department of IT
Vardhaman College of Engineering

01 TOPIC Inheritance Basics

02 TOPIC Inheritance types and its program

03 TOPIC super Keyword

04 TOPIC Method Overriding, Dynamic Method Dispatch

05 TOPIC Abstract classes, final keyword

06 TOPIC Defining a Package, Finding Packages and Class path

07 TOPIC Access Protection, Importing Packages

08 TOPIC Defining and Implementing interfaces, Extending interfaces

Inheritance

- Inheritance in java is a mechanism in which **one object acquires the properties and behaviors of another object.**
- The **idea behind inheritance** in java is that we can **create new classes that are built upon existing classes.**
- When we **inherit from an existing class**, we can **reuse methods & data of parent class**;
- Inheritance represents the **IS-A relationship**, also known as **parent-child relationship.**
- Need of inheritance in java
    1. **For Method overriding (so runtime Polymorphism can be achieved).**
    2. For **Code Reusability**
    3. **It allows creation of Hierarchical Classification.**

Syntax of Inheritance

```
class Subclass-name extends Superclass-name
{
                //methods and variables of sub class
}
```

- The **extends keyword indicates** that we are making a **new class that derives from an existing class.**
- A **class which is inherited** is called **parent or super class.**
- The **class that does the inheriting** is called **child or subclass**.
- A **subclass is a specialized version** of a **super class.**

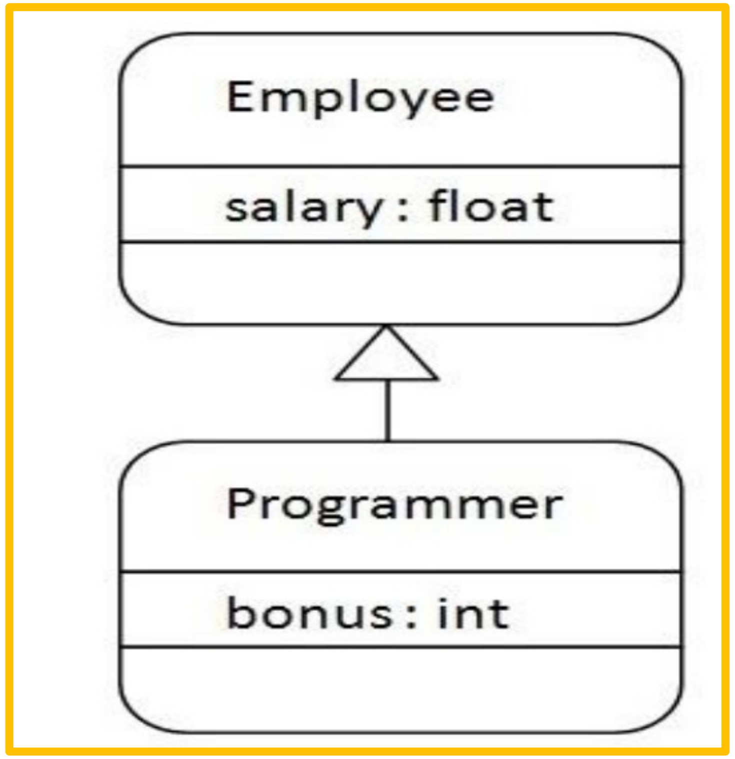
Example:

✓ A Student Is-A Person

✓ Programmer Is-A Employee

Example

- Here **Programmer** is the **subclass** and **Employee** is the **super class.**
- **Relationship** between **two classes** is Programmer **IS-A** Employee.
- It means that **Programmer** is a type of **Employee.**



Example

```
class Employee
 {
       float salary=40000;
 }
 class Programmer extends Employee
 {
       float bonus=(0.2)*salary;
 }

 class TestEmployee
 {
       public static void main(String args[])
       {
           Programmer p=new Programmer();
           System.out.println("Programmer salary
       is:"+p.salay);
             System.out.println("Bonus of Programmer
       is:"+p.bonus);
       }
 }
```

OUTPUT

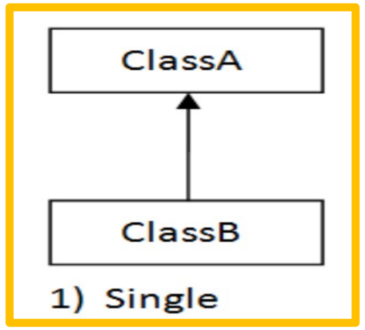
```
Programmer salary is: 40000.0
Bonus of programmer is: 10000
```

In the above example,

- ✓ Programmer object can access the field of own class as well as of Employee class

Types of inheritance

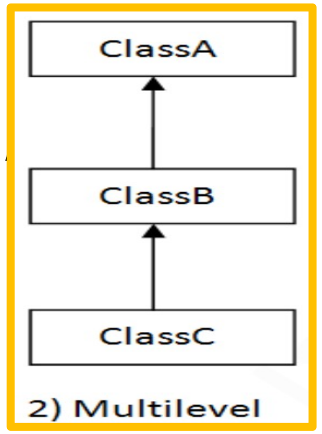
On the basis of class, there can be **three types of inheritance** in **java**:



i.Single Level Inheritance:

A **super class is inherited by only one sub class.**

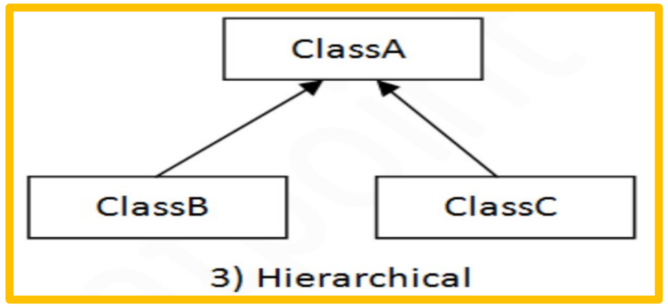
ii.Multi Level Inheritance:



A **sub-class will be inheriting parent class** and as well as **ass act as a parent class** to **other class**., and so on.

iii.Hierarchical Inheritance:

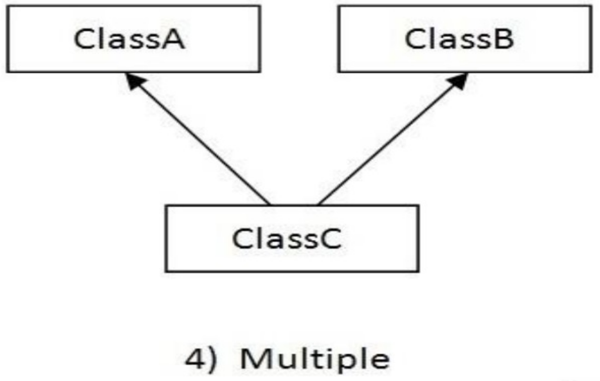
A **Super class is inherited** by **many sub**



Types of inheritance

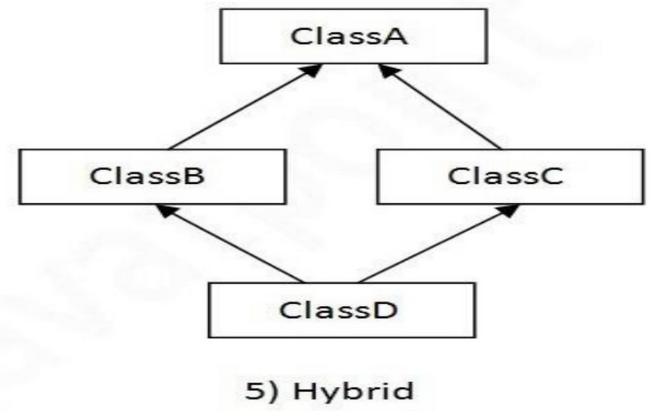
iv.Multiple Inheritance:

A **sub class extending more than one**



v.Hybrid Inheritance:

It is a **combination of Multiple and Multilevel inheritance**



Note:

- **Java does not support Multiple Inheritance** and **Hybrid Inheritance** directly with classes.
- In Java **multiple and hybrid inheritance** is **supported through interfaces only.**

```
i.e : class C extends A,B
 {
          //ComplieTimeError
```

//Example program for Single Level Inheritance

```
class Bird
{
      void fly()
      {
      System.out.println("I am a Bird");
      }
}
class Parrot extends Bird
{
      void colour()
      {
      System.out.println("I am green!");
      }
}
class Test
{
      public static void main(String args[])
      {
      Parrot obj = new Parrot();
      obj.colour();
      obj.fly();
      }
}
```

OUTPUT:

```
I am green!
I am a Bird
```

//Program to Demonstrate Single Level Inheritance

```
class A
{
 void showA()
 {
 System.out.println("show method of ClassA");
 }
}
class B extends A
{
 void showB()
 {
 System.out.println("show method of ClassB");
 }
}
public class SingleLevel
{
 public static void main(String args[])
 {
 B b = new B();
 b.showA();
 b.showB();
 }
}
```

OUTPUT:

```
show method of ClassA
show method of ClassB
```

//Program for Multi Level Inheritance

```
class Bird
{
	void fly()
	{
	System.out.println("I am a Bird");
	}
}
class Parrot extends Bird
{
	void colour()
	{
	System.out.println("I am green!");
	}
}
class SingingParrot extends Parrot
{
	void sing()
	{
	System.out.println("I can sing");
	}
}
```

```
class Test
{
	public static void main(String args[])
	{
	SingingParrot obj = new
SingingParrot();
	obj.sing();
	obj.colour();
	obj.fly();
	}
}
```

OUTPUT:

```
I can sing
I am green!
I am a Bird
```

//Program for Multi Level Inheritance

```
class A
{
      int i=10;
      void showA()
      {
      System.out.println(" show() method of A"+i);
      }
}
class B extends A
{
      int j=25;
      void showB()
      {
      System.out.println(" show() method of B"+j);
      }
}
class C extends B
{
      int k=20;
      public void showC()
      {
      System.out.println(" show() method of C"+k);
      System.out.println("The members sum is " + (i +j
+k));
      }
```

```
class MultiLevel
{
public static void main(String
args[])
{
C c = new C();
c.showA();
c.showB();
c.showC();
 }
}
```

OUTPUT:

```
    show() method of A 10
    show() method of B 25
    show() method of C 20
   The members sum is 55
```

Inheritance

```
class Bird
{
     void fly()
     {
       System.out.println("I am a Bird");
     }
}
class Parrot extends Bird
{
     void colour()
     {
                System.out.println("I am
green!");
     }
}
class Crow extends Bird
{
     void colour()
     {
                System.out.println("I am
black!");
     }
```

```
class Test
{
     public static void main(String args[])
     {
          Parrot p = new Parrot();
          Crow c = new Crow();
          p.colour();
           p.fly();
           c.colour();
          c.fly();
     }
}
```

OUTPUT:

```
I am green!
I am a Bird
I am black!
I am a Bird
```

//Program for Hierarchical Inheritance

```
class A
{
       int i=10;
       void showA()
      {
        System.out.println(" show() 
method of A"+i);
      }
}
class B extends A
{
        int j=25;
        void showB()
        {
         System.out.println(" show() 
method of B"+j);
        }
}
```

```
class C extends A
{
             int j=99;
             void showC()
             {
      System.out.println(" show() method of 
C"+j);
              }
}
class HierLevel
{
      public static void main(String args[])
      {
             B b = new B();
             b.showA();
             b.showB();
             C c = new C();
             c.showA();
             c.showC();
      }
}
```

OUTPUT:

```
show() method of
         A 10
show() method of
         B 25
show() method of
         A 10
show() method of
         C 99
```

super keyword

- The purpose of **“super”** keyword is

  i. **To call superclass methods** and **constructor** fro

Syntax
super (arguments list);

ii. If a **subclass has** the **same member** as **super class**, the **subclass hides** the **members of super class. To** **access** the **super class member** in **subclass** uses **super keyword.**

✓ **super keyword** is to **eliminate** the **confusion between superclasses** and **subclass** n the **s**

Syntax
super.member or
super.method()

Example
super.i;
super.show();

iii.super can be used to **refer immediate super class instance variable**, **method** and **constructor.**

- **super must be** the **first statement** in the **sub class constructor.**
- **super ()** is provided by **compiler implicitly** as **default constructor** if no

```
// super() is provided by the compiler
implicitly
 class Animal
 {
      Animal()
     {
          System.out.println("animal is created");
     }
 }
class Dog extends Animal
 {
     Dog()
     {    // super();                calls
automatically.
          System.out.println("dog is created");
     }
 }
class Test
 {
     public static void main(String args[])
     {
          Dog d=new Dog();
     }
```

Output

```
animal is created
dog is Created
```

constructor execution

```
 class A
{
     A()
     {
     System.out.println("Inside A's
constructor");
     }
 }
 class B extends A
 {
     B()
      {
     System.out.println("Inside B's
constructor");
     }
 }
 class C extends B
 {
     C()
     {
      System.out.println("Inside C's
constructor");
```

```
 class SuperDemo4
 {
     public static void main(String
args[])
     {
          C c = new C();
     }
 }
```

Output

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

// Accessing data members (same name Varibles) of super class

```
class Animal
{
      String name="This is Animal";
}

class Lion extends Animal
{
      String name="This is Lion";
      void display()
      {
       System.out.println(name);
      System.out.println(super.name);
      }
}
class Test
{
      public static void main(String[] args)
      {
        Lion  L= new Lion();
        L.display();
      }
}
```

Output

```
This is
Animal
This is Lion
```

// Accessing methods (same name methods) of super class

```
class Parent
{
      void show()
      {
      System.out.println("Parent Method");
      }
}
class Child extends Parent
{
      void show()
      {
      super.show();
      System.out.println("Child Method");
      }
}
class Test
{
      public static void main(String[] args)
      {
       Child c = new Child();
      c.show();
      }
```

Output

```
Parent
Method Child
Method
```

Method Overriding

- If **subclass has** the **same method name and signature as** a **method in its super class**, it is known as **method overriding in Java**.
- In **that case subclass** will **hide** the **super class method.**
- Rules for Method Overriding
  i. **Method in sub class** must have **same name** and **parameter signature** as **super class.**
  ii. Must be **Is-A** relationship. (Inheritance).

Method Overriding Example

```
{
     int i,j;
     A(int a , int b)
     {
     i=a;
     j=b;
     }
     void show()
     {
     System.out.println("Value of i
is:"+i);
     System.out.println(("Value of i
is:"+j);
     }
 }
class B extends A
{
     int k;
      B(int a , int b , int c)
     {
     super(a,b);
     k=c;
     }
```

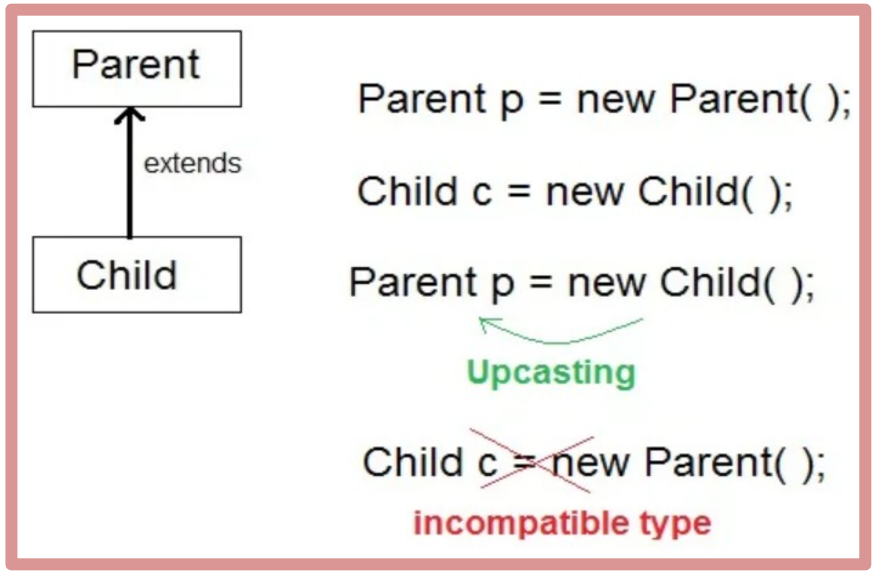
```
     void show()
     {
      super.show();
      System.out.println ("Value of k is:"+k);
     }
}

class Test
{
     public static void main(String args[])
     {
     B b = new B(10,20,30);
     b.show();

     }
}
```

Runtime Polymorphism (Dynamic Method Dispatch)

- Using **Dynamic Method Dispatch**, a **call** to an **overridden method** is **resolved** at **runtime**, **rather than at compile time.**
- **Runtime polymorphism** in java **implemented** by using **Dynamic method dispatch.**
- Upcasting: A **super class reference variable** can **refer** to a **sub class object**. I.e. When reference variable of super class refers to the sub class object, then it is called **up casting.**



```
Parent p = new Parent( );
Child c = new Child( );
Parent p = new Child( );
Child c = new Parent( );
```

// Dynamic Method Dispatch – Example1

```
class Game
{
	void type()
	{
	System.out.println("Indoor  and outdoor");
	}
}

Class Cricket extends Game
{
	void type()
	{
	System.out.println(“It is outdoor game");
	}
}
```

```
Class Test
{
public static void main(String[] args)
	{
	Game gm = new Game();
	Cricket ck = new Cricket();
	gm.type();
	ck.type();
	gm = ck;		//gm refers to Cricket
object
	gm.type();	//calls Cricket's version of
type
	}
}
```

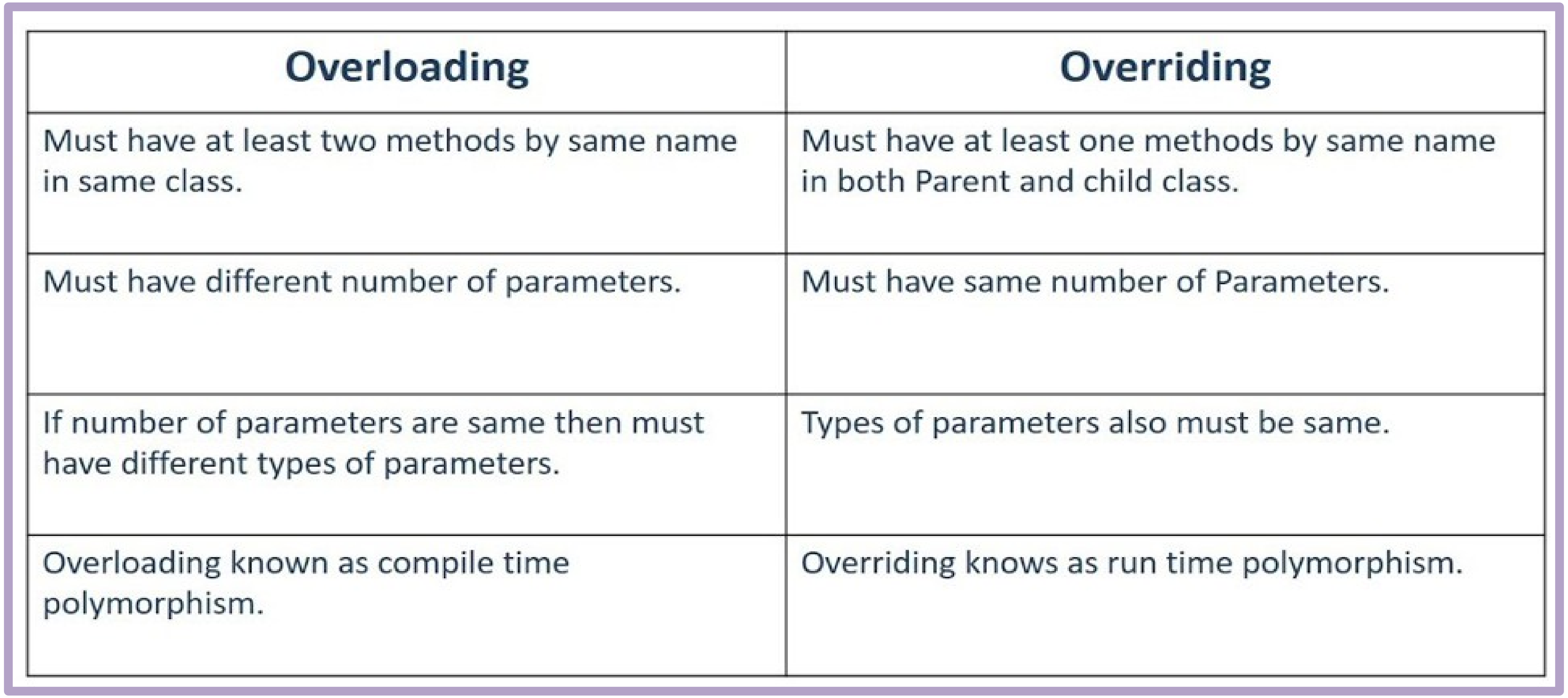
Output

```
Indoor  and outdoor
It is outdoor game
It is outdoor game
```

```
class Bank
 {
      float Interest()
      {
          return 0;
      }
 }
 class SBI extends Bank
 {
      float Interest()
      {
          return 8.4f;
      }
 }
 class ICICI extends Bank
 {
      float Interest()
       {
           return 7.3f;
      }
 }
```

```
class AXIS extends Bank
 {
      float Interest()
       {
           return 9.7f;
      }
 }
class Test
 {
      public static void main(String args[])
      {
             Bank b;
             b=new SBI();
             System.out.println("SBI Rate of Interest:
"+b.Interest());
             b=new ICICI();
             System.out.println("ICICI Rate of Interest:
"+b.Interest());
             b=new AXIS();
             System.out.println("AXIS Rate of Interest: "+b.
Interest());
       }
 }
```

Differentiate method overloading and method overriding



Overloading	Overriding
Must have at least two methods by same name in same class.	Must have at least one methods by same name in both Parent and child class.
Must have different number of parameters.	Must have same number of Parameters.
If number of parameters are same then must have different types of parameters.	Types of parameters also must be same.
Overloading known as compile time polymorphism.	Overriding knows as run time polymorphism.

Abstract classes

- Abstraction is a **process of hiding implementation details** and **showing only functionality** to the **users.**

Eg: Sending a Message, Driving a car etc.

- Abstraction in java achieved using :
  - ✓ **Abstract classes (0 to 99.9%)**
  - ✓ **Interfaces (100%)**

Abstract classes

- **Creating a super class** that **only defines** a **method declaration** that will be **shared by all of its sub classes**; the **sub classes** must **provide implementation**.
- The **method which** is **implements** by the **sub class must be specified as "abstract"**.

Abstract classes

- An **abstract class** is **non-concrete class.**
- **Concrete class** is a **class that has an implementation** for **all** of its **methods**. They **cannot have any unimplemented methods**. It is a **complete class** and **can be instantiated.**
- **Non-Concrete class** is a class that **has no implementation(no body)** for **some of the methods** defined in it. It is an **incomplete class.**

Abstract method Syntax

```
abstract   returntype
methodName(arguments);
```

Abstract class Syntax

```
abstract class class_name
    {
            abstractMethod();
            normalMethod()
            {
                    #body of the method
            }
    }
```

Rules for abstract modifier

- **All abstract methods** of super class must be **implemented** in **subclass** by **overriding.**
- If any **abstract method** is **not implemented** in **sub class**, then the **sub class** is also **made as abstract class.**
- **No objects** of an **abstract class exist**. i.e. An **abstract class cannot be directly instantiated** with the **new operator.**
- An **abstract class not fully defined**. Hence **object cannot be created.**
- **Abstract classes** can be **used to create object references** because of **Dynamic Method Dispatch** and **Run time polymorphism**

```
abstract class Biggboss
{
     abstract void season();
     abstract void winner();
     void runner()
     {
       System.out.println("The runner is: Srihan");
     }

}
class starmaa extends Biggboss
{
     void season()
     {
       System.out.println("This is Biggboss
Season 6");
     }
     void winner()
     {
       System.out.println("The winner is:
Revanth");
     }
  }
```

```
class BB6
{
     public static void main(String
args[])
     {
               starmaa s = new
starmaa();
          s.season();
          s.winner();
          s.runner();
     }
}
```

Output

```
This is Biggboss Season
6
The winner is: Revanth
The runner is: Srihan
```

Abstract classes

```
abstract class Bike
{
     abstract void run();
     void engine()
     {
           System.out.println(“Bike engine");
     }
}
class Honda extends Bike
{
     void run()
     {
          System.out.println(“Running safely");
     }
}
```

```
Class Test
{
     public static void main(String
args[])
     {
          Bike B = new Honda();
          B.run();
          B.engine();
     }
}
```

Output

```
Running
safely
Bike engine
```

Abstract classes

```
abstract class Calculator
{
    abstract void display();
}

class Add extends Calculator
{
    void display()
    {
        System.out.println(“This is
Addition class");
    }

}
```

```
class Sub extends Calculator
{
    void display()
    {
        System.out.println(“This is
Subtraction Class");
    }

}
class Test
{
    public static void main(String arg[])
    {
        Calculator c1 = new Add();
        c1.display();
        Calculator c2 = new Sub();
        c2.display();
    }
}
```

Output

```
This is Addition class
This is Subtraction
Class
```

Abstract classes

```
abstract class Shape
{
	abstract void draw();
}
class Rectangle extends Shape
{
	void draw()
	{
	System.out.println("drawing 
rectangle");
	}
}
class Circle extends Shape
{
	void draw()
	{
	System.out.println("drawing circle");
	}
}
```

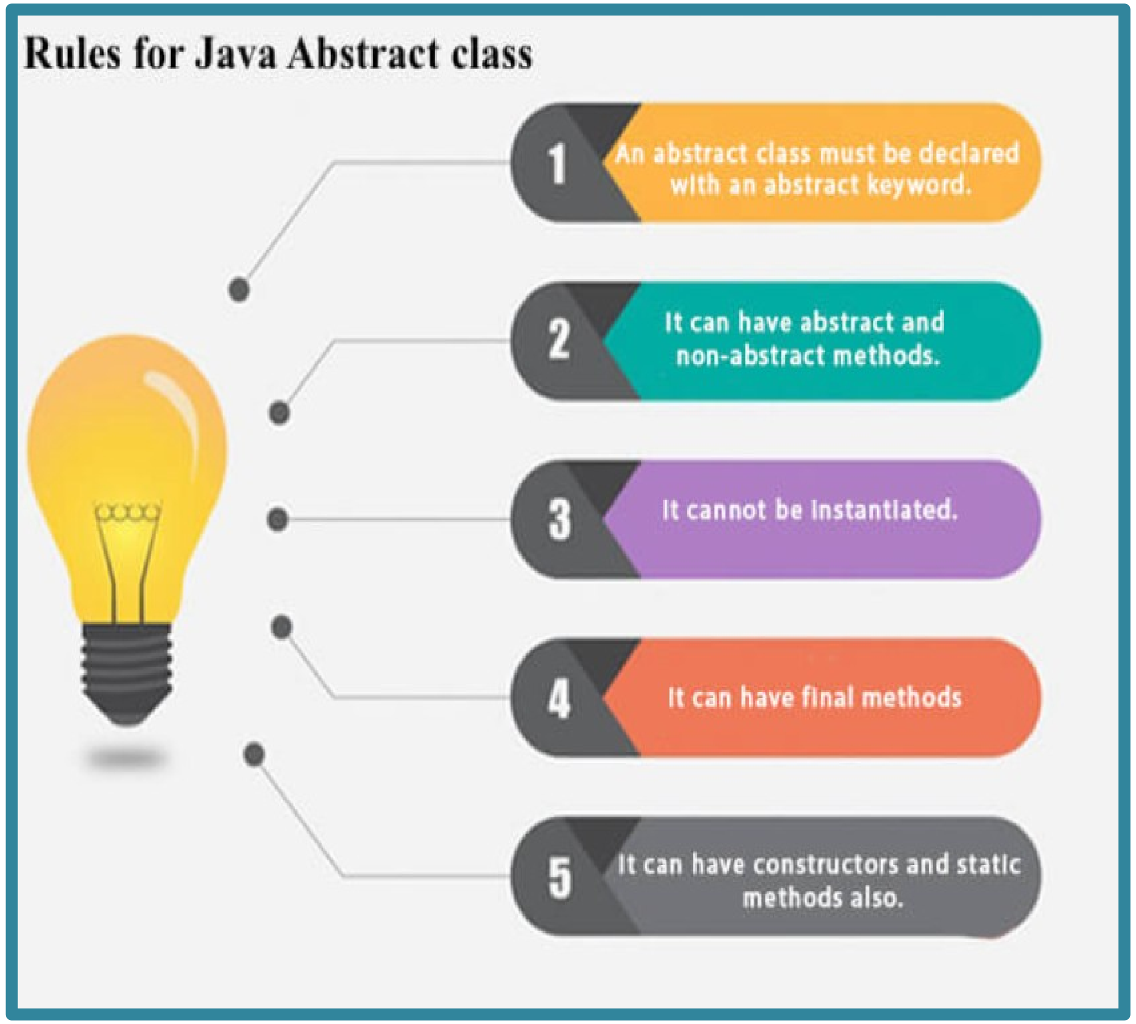
```
class Test
{
	public static void main(String 
args[])
	{
	Shape s1=new Circle();
	s1.draw();
	Shape s2=new Rectangle();
	s2.draw();

	}
}
```

Output

```
drawing circle
drawing rectangle
```

Rules for abstract class


Rules for Java Abstract class

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have final methods
5. It can have constructors and static methods also.

final keyword

- The **final keyword** in java is used to **restrict the user.**
- final is used with a **variable, method , or a class**.

i.final keyword with a variable

- A **variable declared** as **final prevents its contents** from **being modified**.(Constant)
- We **must initialize a final variable when it is declared.**
- The **value** can be **initialized during declaration** or in the **constructor.**
- A **local variable** or parameter **can also be made as “final”**

```
//Example to demonstrate final
variable
class test
{
  public static void main(String[]
args)
{
    final int AGE = 22;
    AGE = 25;
    System.out.println("Age: " + AGE);
 }
}
```

final keyword with a method

```
class Bike
 {
     final void show()
      {
     System.out.println("Bike Method");
     }
 }

class Honda extends Bike
 {
     //inherits show()
     void show()
     {
     System.out.println("Honda's Show");
     }
     void display()
      {
      System.out.println("Honda's
Show");
     }
 }
```

A **method declared as final**, will **not allow overriding**. i.e. It prevents overriding / stops overriding.

A **constructor cannot** be **declared as final.** Because it is never inherited.

A **final method is inherited**, **but not**

```
class Test
 {
     public static void main(String args[])
      {
     Honda h1 = new Honda();
     h1.display();
     h1.show();          //compile time error
     }
 }
```

final keyword with a class

- A class declared as final will not allow extending. i.e. It stops inheritance.

```
final class Bike
 {
     final void show()
     {
     System.out.println("Bike's Method");
     }
}

class Honda extends Bike //Compile time
error
 {
     //trying to inherit - CTE
 }
```

Conclusion of Final keyword

- ✓ **Stop Value Change**
- ✓ **Stop Method overriding**
- ✓ **Stop inheritance**

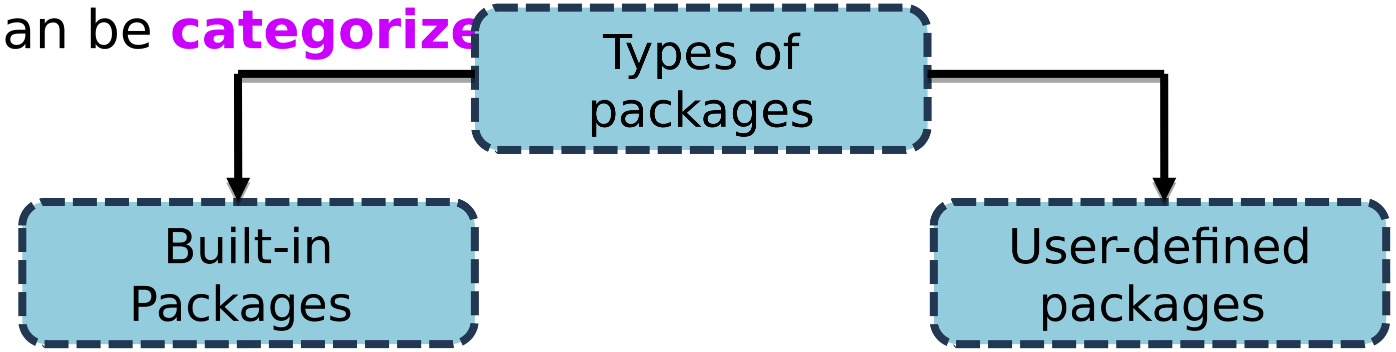
Is final method inherited?

- If we **declare a parameter as final**, it **cannot allow changing in its method.**

```
class Bike
{
       final void run()
      {
      System.out.println(“Bike running");
      }
}
class Honda extends Bike
{
      public static void main(String args[])
      {
      Honda h=new Honda();
      h.run();
      }
}
```

Packages

- Package in Java is a **mechanism to encapsulate** a **group of classes**, **sub packages** and **interfaces**. **Packages** are **containers** for **classes.**
- Packages are **stored in hierarchical manner** and are **explicitly imported** into **new classes**.
- Package in java can be **categorize**



1.Built-in Packages :

✓ These **packages consist** of a **large number of classes** which are a **part of Java API**. Some of the **commonly used built-in packages are:**

i. java.lang: Contains **language support classes** (Integer, Throwable, Threadetc).

Introduction to Packages

ii. java.io: Contains classed for **supporting input / output operations.**

iii. java.util: Contains utility classes which implement data structures like **Scanner**, **Collections, Date, StringTokenizer** etc.

2. User-defined packages

- ✓ These are the **packages that are defined by the user.**

Advantages of Java Packages

- ✓ We can **reuse existing classes** from the packages **as many times as we need it in our program**.
- ✓ Java Package **provides access protection.**
- ✓ Java Packages **removes naming collision.**
- ✓ Packages can be **considered as data encapsulation.**
- ✓ Packages can be **organized** as **hierarchical structures** with **sub packages and classes.**

Defining a Package

- The **"package" keyword** is used to **create** a **package** in **java.**
- A package is defined using **"package" command** as the **first statement** in the **java source file.**
- The "**package**" statement defines a **name space in which** the **classes** are **stored.**
- The general forr

Syntax

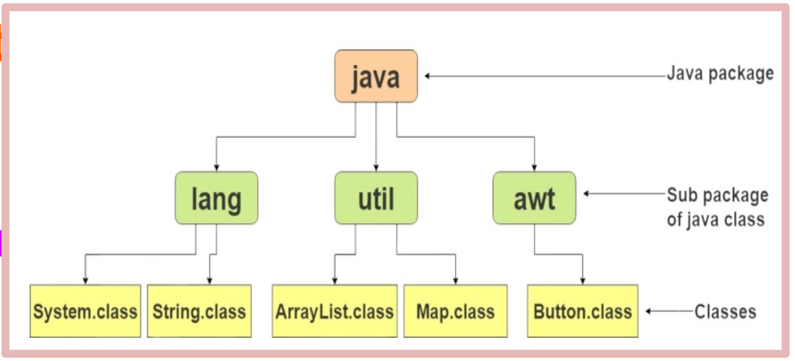
```
package
pkgname;
```

Example

```
package
mypack;
```

- Java **uses file system directories** to **store the packages**. i.e. **All .class** files must be **kept** in a **directory** called **mypack.**
- **More than one file** can include the **same package stat**
- We can create a **hierarchy of packages**. Thus forms **mu**
- The **"."** **Operator** is **used** to create **multilevel packag**

Eg: package com.vce.cse



Importing Packages

- The "**import**" statement in java allows **accessing a package.**
- The form of "import" statement is:

Syntax-1

```
import pkg1.pkg2. .....
pkgn;
```

Example-1

```
import
java.util.Scanner;
```

Syntax-2

```
import
pkg1.*;
```

Example-2

```
import
java.util.*;
```

Note:After creation of Package program,we need compile it by using below syntax:

Syntax-1

```
>javac –d .
Classname.java
```

Example

```
>javac –d . Animal.java
```

–d is used to save the class file in the directory and the '.' (dot) denotes the package in the current directory. To avoid name conflicts, please use lower case for package names.

Example-1

//Program to Demonstrate package

```
package Animals;
public class PetAnimals
{
     public void Dog()
     {
     System.out.println("This is Dog");
     }
     public void Cat()
     {
     System.out.println("This is Cat");
     }
}
```

Note:
‘PetAnimals .class’ file is stored in’ Animals’ folder.

```
package Animals;
public class WildAnimals
{
     public void Lion()
     {
     System.out.println("This is Lion");
     }
     public void Tiger()
     {
     System.out.println("This is Tiger");
     }
}
```

Note:
WildAnimals.class file is stored in Animals folder.

Example-1

```
//Program for Accessing Package

import Animals.PetAnimals;
import Animals.WildAnimals;
class Test
{
	public static void main(String args[])
	{
		PetAnimals p=new PetAnimals();
		p.Dog();
		p.Cat();
		WildAnimals w=new WildAnimals();
		w.Lion();
		w.Tiger();
	}
}
```

package

```
package mypack;
public class Calculator
{
     public int add(int a , int b)
     {
          return(a+b);
     }
     public int sub(int a , int b)
     {
          return(a-b);
     }
     public int mul(int a , int b)
     {
          return(a*b);
     }
     public int div(int a , int b)
     {
          return(a/b);
     }
}
```

Calculator.class file is stored in mypack

Example-2

```
package mypack;
public class Factorial
 {
     int n;
     public Factorial(int n)
     {
          this.n = n;
     }
     public int fact()
      {
          int f=1;
          for(int i=1;i<=n;i++)
               f=f*i;
          return f;
     }
}
```

Factorial.class file is stored in mypack folder.

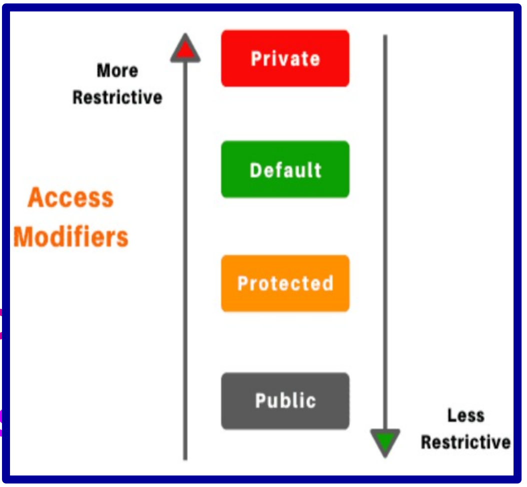
Example-2

```
//Accessing Package
import mypack.Calculator;
import mypack.Factorial;
public class PkgDemo
{
    public static void main(String args[])
    {
    Calculator c = new Calculator();
    int x=10;
    int y=20;
    System.out.println("Addition" + c.add(x,y));
    System.out.println("Multiplication" +
c.mul(x,y));
    Factorial f = new Factorial(6);
    int res = f.fact();
    System.out.println("The factorial is " +res);
    }
}
```

Access Modifiers In Java

- These are **powerful tools** help you determine **who can use or modify different parts of your code**, helping to **keep your projects organized** and **secure.**
- **Access modifiers** are **keywords** that can be **used to control the accessibility** of **fields, methods, and constructors in a class.**
- The four access modifiers in Java are:

public, protected, default, and private



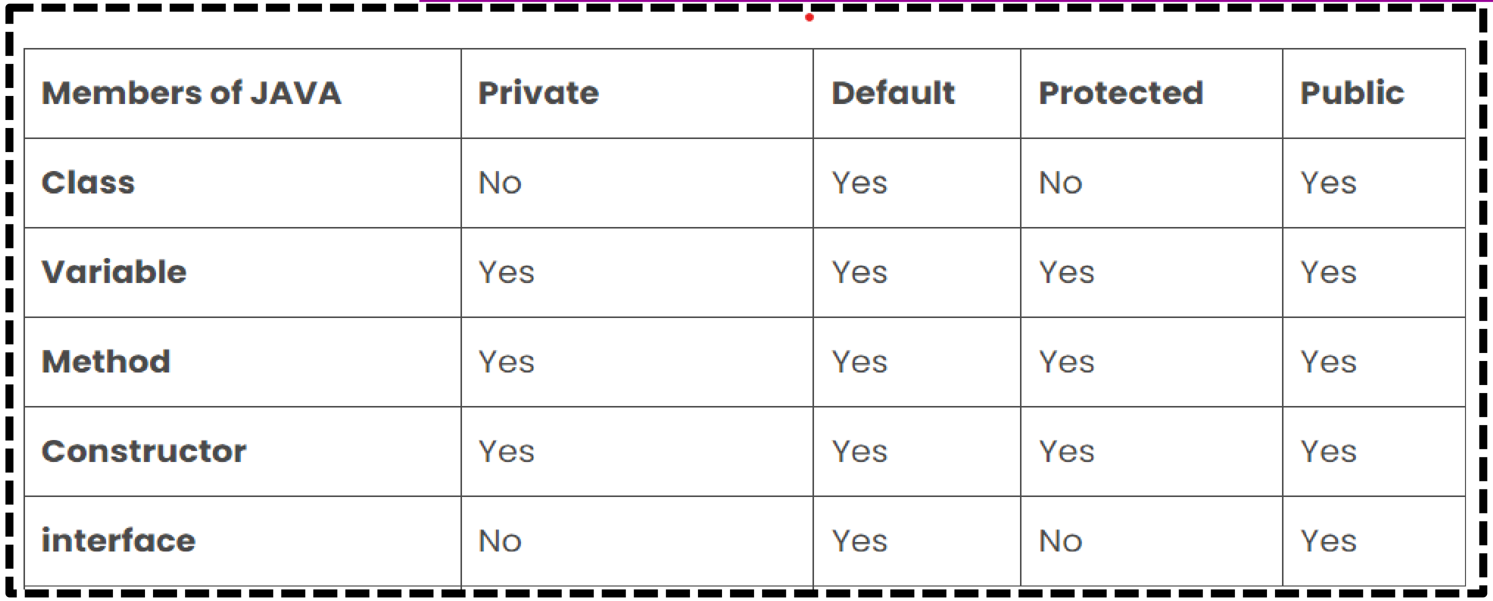
i. public: **Anything declared public** can be **accessed fro**

ii. private: Anything declared **private cannot be seen out**

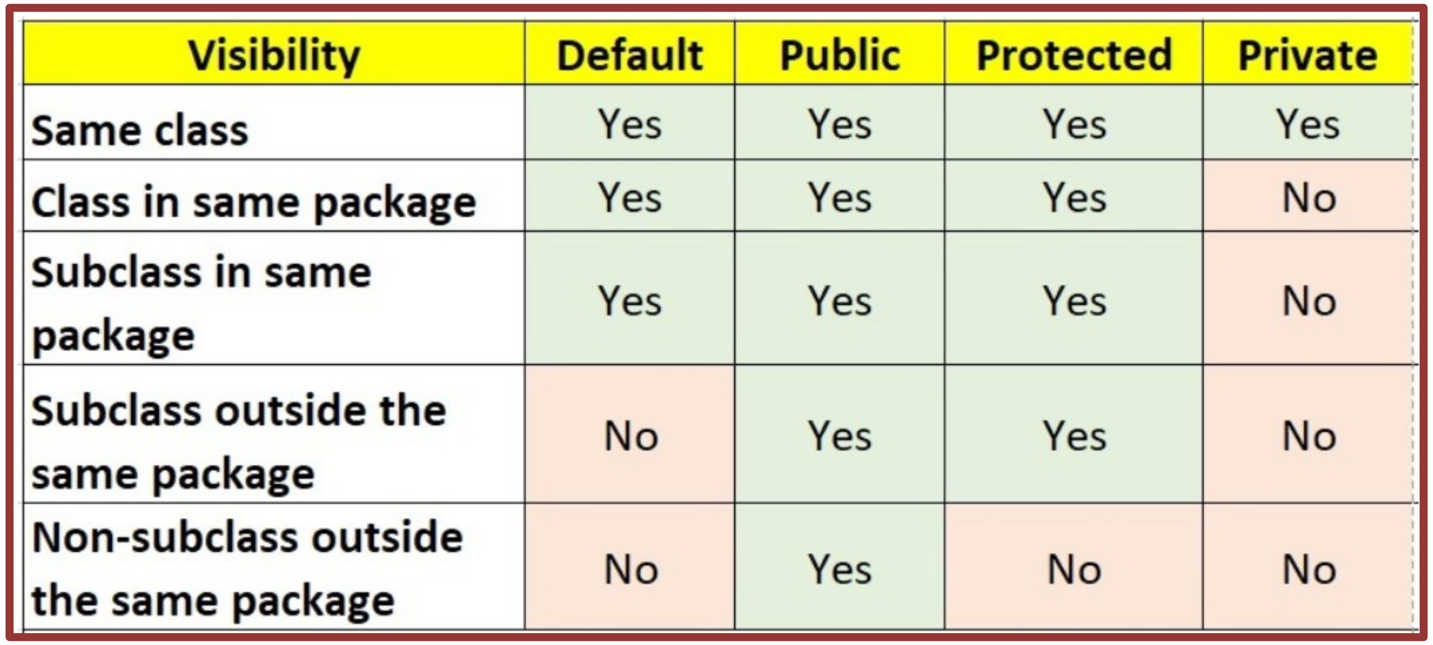
iii. default: When a member **does not have an explicit access specification**, it is **visible to subclasses** as well as to **other classes in the same package.** This is the default access.

iv. protected: The access level of a protected modifier is **within the package** and **outside the package through child class.** If you do not make the child class,

Access Modifiers In Java



Members of JAVA	Private	Default	Protected	Public
Class	No	Yes	No	Yes
Variable	Yes	Yes	Yes	Yes
Method	Yes	Yes	Yes	Yes
Constructor	Yes	Yes	Yes	Yes
interface	No	Yes	No	Yes



Visibility	Default	Public	Protected	Private
Same class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in same package	Yes	Yes	Yes	No
Subclass outside the same package	No	Yes	Yes	No
Non-subclass outside the same package	No	Yes	No	No

//Accessing a private data and method from same class

```
class college
{
	private String Name="Vardhaman";
	private void show()
	{
	System.out.println("I am private function");
	}
	public static void main(String[] args)
	{
	college c=new college();
	c.show();
	System.out.println(c.Name);
	}
}
```

Output

```
I am private
function
Vardhaman
```

//Accessing a private data and method from different class

```
class college
{
	private String Name="Vardhaman";
	private void show()
	{
	System.out.println("I am private function");
	}
}
Class Test
{
	public static void main(String[] args)
	{
	college c=new college();
	c.show();
	System.out.println(c.Name);
	}
}
```

Output

```
Compile Time
Error
```

```
//Accessing a protected data and  methods
class A
{
   protected void display()
    {
        System.out.println("I am protected function");
    }
}
class B extends A {}
class C extends B {}

class Test
{
     public static void main(String args[])
   {
       B obj1 = new B();
       obj1.display();
       C obj2 = new C();
       obj2.display ();
   }
}
```

Output

```
I am protected
function
I am protected
function
```

Interfaces

- The **keyword 'interface'** **used** to **defining** an **interface.**
- An Interface **allows creating** a **fully abstract class.**
- An **interface specifies** what **a class must do** but **not how it does it.**
- An Interface has **methods declared without any body** and it can have **Final and static variables.**
- **Once an interface is defined**, **any number of classes** can **implement an interface.**
- The **class which implements** an **interface must provide definition** for **all the methods** of an **interface.**
- It is **used to achieve abstraction** and **multiple inheritance in Java**.
- Java **Interface** also **represents IS-A relationship.**
- It **cannot be instantiated** just **like abstract class.**
- To **implement** an **interface**, **use "implements"** **clause** in the **class definition .**

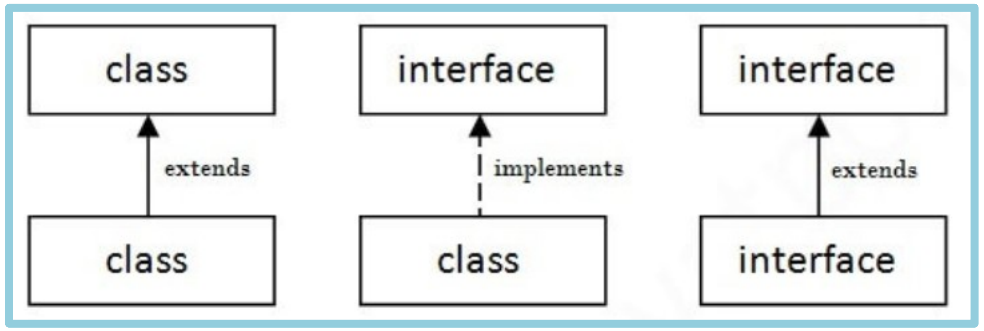
Syntax for Defining Interface

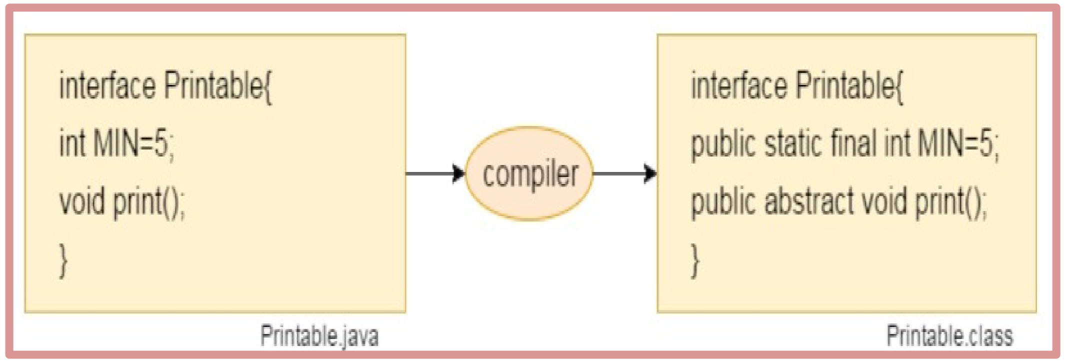
```
interface InterfaceName
{
    returntype
methodName1(Arguments);
    returntype
methodName2(Arguments);
    ------------------------------
    ------------------------------
     ------------------------------
    Datatype  variable1 = value;
    Datatype  variable1 = value;
    ------------------------------
     ------------------------------

}
```

Syntax for implementation of Interface

```
Class  className implements
interface1 , interface2
 {
    ----------------
    Class body
     ----------------
 }
```





```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

```
interface Language
{
     void Name();
     void type();
     void version();
}
class Proglan implements Language
{
     public void Name()
     {
       System.out.println("I am JAVA");
     }
     public void type()
     {
       System.out.println("I am Programming
Language");
     }
     public void version()
     {
       System.out.println("My Latest version
20.0");
     }
```

```
class Test
{
     public static void main(String
args[])
     {
                Proglan j = new Proglan();
          j.Name();
          j.type();
          j.version();
     }
}
```

Output

```
I am JAVA
I am Programming
Language
My Latest version 20.0
```

```
{
     void Dog();
     void Cat();
}
Interface WildAnimals
{
     void Lion();
     void Tiger();
}
class Animals implements
PetAnimals,WildAnimals
{
     void Dog()
     {   System.out.println(“This is Dog");
}
     void Cat()
     {  System.out.println(" This is Cat
");  }
     void Lion()
     {  System.out.println(" This is
Lion");  }
     void Tiger()
     { System.out.println(" This is
```

```
class Test
{
     public static void main(String args[])
     {
               Animals A = new Animals();
          A.Dog();
          A.Cat();
          A.Lion();
          A.Tiger();
     }
}
```

Output

```
This is Dog
This is Cat
This is Lion
This is Tiger
```

How to extend Interface?

Syntax for extending Interface

```
 interface InterfaceName-1
{
      returntype  methodName-
1(Arguments);
     returntype   methodName-
n(Arguments);
     
}
  interface InterfaceName-2 extends
InterfaceName-1
{
      returntype  methodName-1(Arguments);
     returntype   methodName-m(Arguments);
     
}
Class implements InterfaceName-2
{
     --------------------------
     --------------------------
}
```

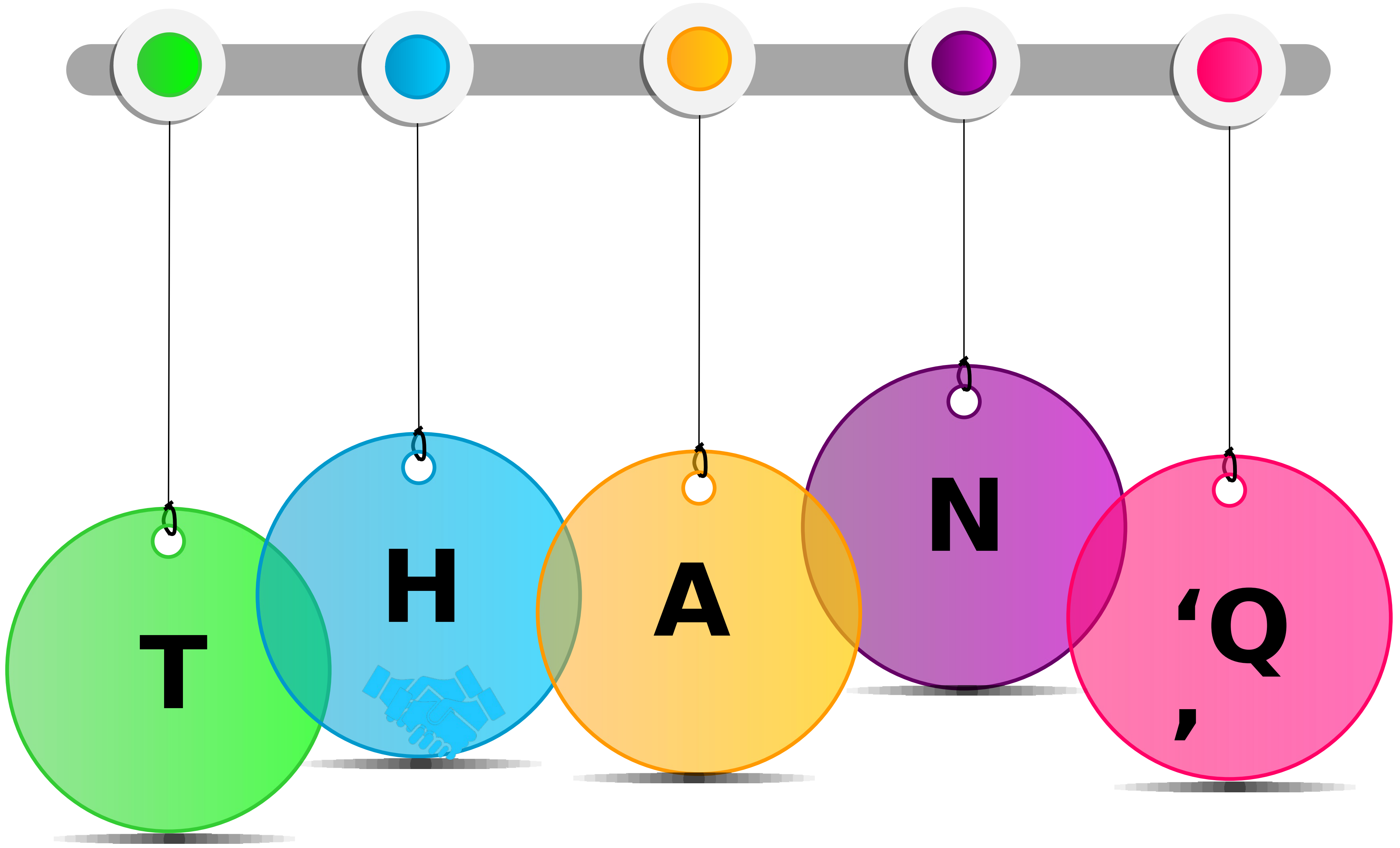
//Program for extending Interface

```
interface Books
{
      void Notebooks();
      void Textbooks();
 }
Interface Storybooks  extends Books
{
      void Title();
}
class Booktype implements Storybooks
{
       void Notebooks()
      {    System.out.println("I am Notebook");
      }
       void Textbooks()
      {   System.out.println("I am Textbook");
      }
       void Title()
      {
        System.out.println(“I am Storybook");
      }
}
```

```
class Test
{
      public static void main(String args[])
      {
                  Booktypes B = new Booktypes();
            B. Notebooks();
            B. Textbooks();
            B. Title();
      }
}
```

Output

```
I am Notebook
I am Textbook
I am Storybook
```


T
H
A
N
‘Q
,